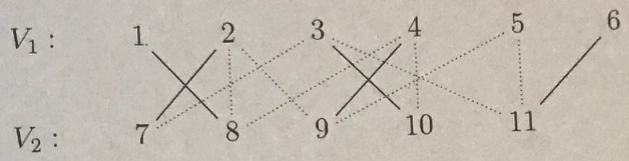
Name: maura kreft
ID: 103947905

CSCI 3104
Problem Set 10

Profs. Grochow & Layer
Spring 2019, CU-Boulder

Quick links 1a 1b 1c 2a 2b 2c 3a 3b

1. A *matching* in a graph G is a subset $E_M \subseteq E(G)$ of edges such that each vertex touches at most one of the edges in E_M. Recall that a bipartite graph is a graph G on two sets of vertices, V_1 and V_2, such that every edge has one endpoint in V_1 and one endpoint in V_2. We sometimes write $G = (V_1, V_2; E)$ for this situation. For example:



The edges in the above example consist of all the lines, whether solid or dotted; the solid lines form a matching.

The *bipartite maximum matching* problem is to find a matching in a given bipartite graph G , which has the maximum number of edges among all matchings in G.

Name: Maura Kieft
ID: 103947905

CSCI 3104 **Profs. Grochow & Layer**
Problem Set 10 **Spring 2019, CU-Boulder**

(a) (6 pts total) Prove that a maximum matching in a bipartite graph $G = (V_1, V_2; E)$ has size at most $\min\{|V_1|, |V_2|\}$.

theorem: a maximum matching in a bipartite graph $G = (V_1, V_2; E)$ has size at most $\min\{|V_1|, |V_2|\}$.

Given any matching M, a vertex cover C must contain at least one of of the endpoints of each edge in M. a vertex cover is a set C of vertices such that all edges $e \in E$ are incident to at least one vertex of C. since no vertex can cover more than one edge, because the edge overlap would prevent M from being a matching in the first place. So, there is no edge completely contained in V−C. Thus, the size of any matching is at most the size of any vertex cover.

proof by contradiction: assuming that $\min\{|V_1|, |V_2|\} = |V_1|$ and that a matching M of G has more edges than $|V_1|$ edges. Given $G = (V_1, V_2; E)$ as a bipartite graph, and matching consists of the edges v_1 to v_2 for matching of G. We assumed M has more than $|V_1|$ edges, and a characteristic of a matching in a bipartite graph is that no two edges can share a common vertex. So, all edges in M must have a different end point in V_1 by definition, which is more than $|V_1|$. which leads us to the conclusion that, since M has more edges than $|V_1|$, there must be at least two vertices that share a common endpoint in M. By definition, this result wouldn't make M a matching.

Thus, our assumption isn't true and, so, M must have at most $|V_1|$ edges for this contradiction, and overall at most $\min\{|V_1|, |V_2|\}$.

Name: maura kieft
ID: 103947905

CSCI 3104 — **Profs. Grochow & Layer**
Problem Set 10 — **Spring 2019, CU-Boulder**

(b) (6 pts total) Show how you can use an algorithm for max-flow to solve bipartite maximum matching on undirected simple bipartite graphs. That is, give an algorithm which, given an undirected simple bipartite graph $G = (V_1, V_2; E)$, (1) constructs a directed, weighted graph G' (which need not be bipartite) with weights $w : E(G') \to \mathbb{R}$ as well as two vertices $s, t \in V(G')$, (2) solves max-flow for $(G', w), s, t$, and (3) uses the solution for max-flow to find the maximum matching in G. Your algorithm may use any `max-flow` algorithm as a subroutine.

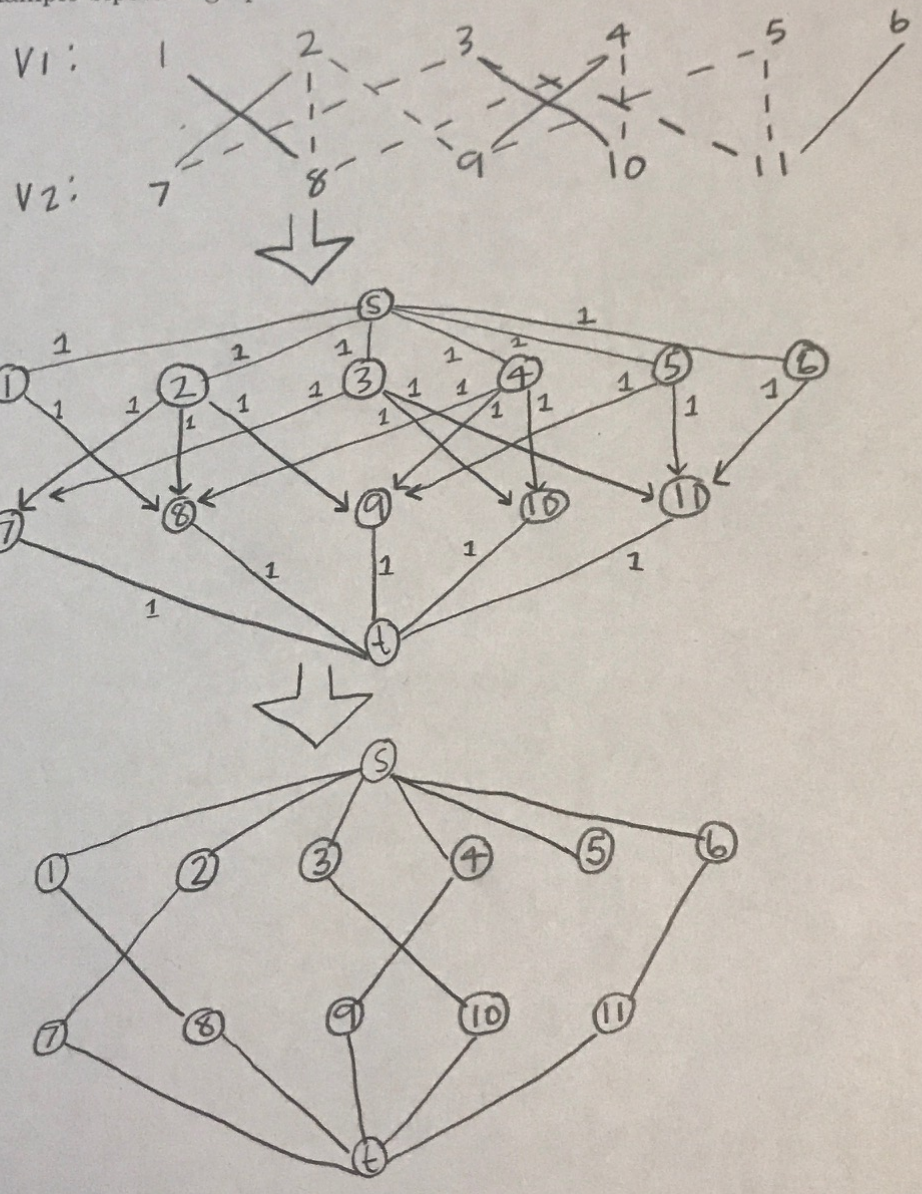
(1) Directed, weighted graph G' with weights $w: E(G') \to \mathbb{R}$ as well as two vertices $s, t \in V(G')$
- Given a bipartite graph $G = (V_1, V_2; E)$, create
  - ↳ create graph by adding the two vertices s and t which will be our source and end (or sink)
  - ↳ add an edge from the vertex s to every vertex in V_1 with capacity as 1 unit
  - ↳ add an edge from every vertex in V_2 to the vertex t with capacity as 1 unit
  - ↳ for every edge in $G = (V_1, V_2; E)$, make the edges directed from V_1 to V_2 and their capacities as 1 unit again.

(2) $(G', w), s, t$
- compute the max-flow from s to t in network in the new, modified, directed graph G', create

(3) maximum matching in G
- edges from V_1 to V_2 after completing max-flow in G' will be a component of matching edges in G.
  - ↳ since edges have capacity as 1 unit and every vertex in V_1 is connected by an edge also with the capacity as 1 unit from source (s) and every vertex in V_2 is connected to the sink (t) also with capacity as 1 unit. so, in order to maximize flow, this algorithm will find the maximum number of vertices paths from source vertex s to sink vertex t, which is also the size of the maximum matching.

(c) (7 pts total) Show the weighted graph constructed by your algorithm on the example bipartite graph above.

Given:



Name: maura Krefft
ID: 103947905

CSCI 3104
Problem Set 10

Profs. Grochow & Layer
Spring 2019, CU-Boulder

2. In the review session for his Deep Wizarding class, Dumbledore reminds everyone that the logical definition of NP requires that the number of *bits* in the witness w is polynomial in the number of bits of the input n. That is, $|w| = poly(n)$. With a smile, he says that in beginner wizarding, witnesses are usually only logarithmic in size, i.e., $|w| = O(\log n)$.

Name: maura Kieft
ID: 103947905

CSCI 3104 **Profs. Grochow & Layer**
Problem Set 10 **Spring 2019, CU-Boulder**

(a) (7 pts total) Because you are a model student, Dumbledore asks you to prove, in front of the whole class, that any such property is in the complexity class P.

Given: logical definition of NP requires that the number of bits in the witness w is polynomial in the number of bits of the input n ($|w| = poly(n)$)
↳ In beginning wizardry, witnesses are usually only logarithmic in size ($|w| = O(\log n)$)

Complexity class P contains problems with efficient algorithms for finding solutions. In regards to complexity class P, the size of an input x is n if it takes n-bits of computer memory to store x, so $|x| = n$. The logical definition of NP requires that the number of bits in the witness w is polynomial in the number of bits of the input n. So, for all inputs $|x| = n$, for P class of decision problems which have polynomial-time algorithms, the number of bits in the witness w is polynomial in the number of the input n based off of the complexity class P. Thus, every problem in P is also in NP since P is a subset of NP. But, we do not know that every problem in NP is also in P. Here we are given that witnesses w is of logarithmic size, so we can conclude that the "logical definition of NP" is able to be reduced to P, and so, this is in class P.

Name: maura Kieft
ID: 103947905

CSCI 3104
Problem Set 10

Profs. Grochow & Layer
Spring 2019, CU-Boulder

(b) (6 pts total) Well done, Dumbledore says. Now, explain why the logical definition of NP implies that any problem in NP can be solved by an exponential-time algorithm.

Since this is in set P, the logical definition of NP states that a requirement is that the number of bits in the witness w is polynomial in the number of bits of the input n. If P is any other problem in NP, then $P \leq_p NP$, and by the theorem (8.1) which says "suppose $P \leq_p NP$, If NP can be solved in polynomial time, then P can be solved in polynomial time." In our case, since $P \leq_p NP$, it implies that since NP can be solved in exponential time, then P can also be solved in exponential time. Hence, $NP \subseteq P$, and thus P (or any problem in NP) can be solved by an exponential-time algorithm since $P \leq_p NP$ such that the value is $P \leq_p NP$.

Name: maura Kieft
ID: 103947905

CSCI 3104
Problem Set 10

Profs. Grochow & Layer
Spring 2019, CU-Boulder

(c) (6 pts total) Dumbledore then asks the class: "So, is NP a good formalization of the notion of problems that can be solved by brute force? Discuss." Give arguments for both possible answers.

Is NP a good formalization of the notion of problems that can be solved by brute force?

(1) Yes, because in this case applying bruteforce on any algorithm should be solved in exponential time because in the algorithm there exists an effective method for deriving the correct answer in which the "verifier" V(X,U) = YES or V(X,U) = NO, making it decidable and verifies if there are answers to the algorim as well as it always terminates giving an output.

(2) NO, because there are problems in complexity class P which will be solved in polynomial time and not take exponential time. so, NP doesn't necessarily formalize problems solvable by brute force since class P contains decision problems which can be solved efficiently by polynomial time algorithms.

Name: maura Kieft
ID: 103947905

CSCI 3104 **Profs. Grochow & Layer**
Problem Set 10 **Spring 2019, CU-Boulder**

3. (20 pts) Recall that the *MergeSort* algorithm is a sorting algorithm that takes $\Theta(n \log n)$ time and $\Theta(n)$ space. In this problem, you will implement and instrument MergeSort, then perform a numerical experiment that verifies this asymptotic analysis. There are two functions and one experiment to do this.

   (i) `MergeSort(A,n)` takes as input an unordered array A, of length n, and returns both an in-place sorted version of A and a count t of the number of atomic operations performed by `MergeSort`.

   (ii) `randomArray(n)` takes as input an integer n and returns an array A such that for each $0 \leq i < n$, $A[i]$ is a uniformly random integer between 1 and n. (It is okay if A is a random permutation of the first n positive integers.)

Name: maura Krefft
ID: 103447905

CSCI 3104
Problem Set 10

Profs. Grochow & Layer
Spring 2019, CU-Boulder

(a) (10 pts total) From scratch, implement the functions `MergeSort` and `randomArray`. You may not use any library functions that make their implementation trivial. You may use a library function that implements a pseudorandom number generator in order to implement `randomArray`.

Submit a paragraph that explains how you instrumented `MergeSort`, i.e., explain which operations you counted and why these are the correct ones to count.

In my implement of mergesort I counted the atomic operations encountered when running the function. atomic operations are simple mathematical and simple functional operations which take O(1) time. So, I counted everytime the function is called, a comparison is made, an array is accessed, or an arithmetic operation is used or encountered I added it the count. These are the correct one's to count based off the definition of atomic operations, simple arithmetic includes addition, multiplication, subtraction, division, equals, etc. and functional operations, such as if comparisons, as well as higher order operations/loops like for, while, etc.

Name: maura kreft
ID: 103947905

CSCI 3104
Problem Set 10

Profs. Grochow & Layer
Spring 2019, CU-Boulder

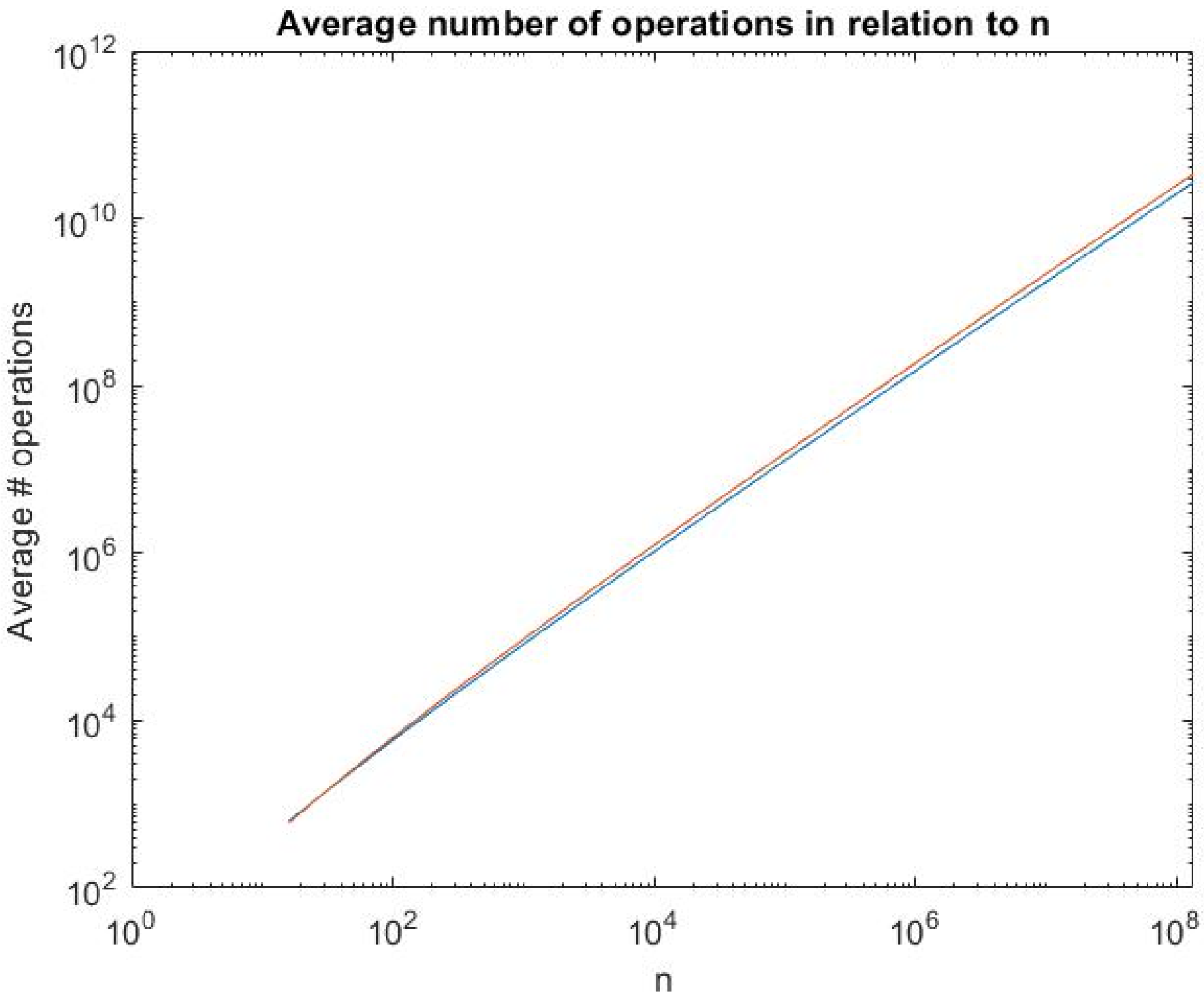
(b) (10 pts total) For each of $n = \{2^4, 2^5, \ldots, 2^{26}, 2^{27}\}$, run `MergeSort(randomArray(n),n)` fives times and record the tuple $(n, \langle t \rangle)$, where $\langle t \rangle$ is the average number of operations your function counted over the five repetitions. Use whatever software you like to make a line plot of these 24 data points; overlay on your data a function of the form $T(n) = A\, n \log n$, where you choose the constant A so that the function is close to your data.

Hint 1: To increase the aesthetics, use a log-log plot.
Hint 2: Make sure that your *MergeSort* implementation uses only two arrays of length n to do its work. (For instance, don't do recursion with pass-by-value.)

$2^4 \to (16, 626)$
$2^5 \to (32, 1482)$
$2^6 \to (64, 3417)$
$2^7 \to (128, 7735)$
$2^8 \to (256, 17268)$
$2^9 \to (512, 38110)$
$2^{10} \to (1024, 83418)$
$2^{11} \to (2048, 181187)$
$2^{12} \to (4096, 391036)$
$2^{13} \to (8192, 839427)$
$2^{14} \to (16384, 1793496)$
$2^{15} \to (32768, 3816499)$
$2^{16} \to (65536, 8091786)$
$2^{17} \to (131072, 17100840)$
$2^{18} \to (262144, 36037296)$

$2^{19} \to (524288, 75744756)$
$2^{20} \to (1048576, 158828684)$
$2^{21} \to (2097152, 332337340)$
$2^{22} \to (4194304, 694035276)$
$2^{23} \to (8388608, 1446787950)$
$2^{24} \to (16777216, 3011018950)$
$2^{25} \to (33554432, 6256915124)$
$2^{26} \to (67108864, 12983603568)$
$2^{27} \to (134217728, 2690671757$

$T(n) = 13.5\, n \log n$


Average number of operations in relation to n
Average # operations
n
10^{12}
10^{10}
10^{8}
10^{6}
10^{4}
10^{2}
10^{0}
10^{2}
10^{4}
10^{6}
10^{8}